주기억장치 데이터베이스 기반 트랜잭션 처리 시스템의 설계 및 평가

심 종 익[†]

요 약

최근 들어 신속한 트랜잭션 처리를 요구하는 데이터베이스 응용이 확대되고 있다. 트랜잭션 처리 시스템에서 높은 성능을 달성하기 위한 한가지 방법으로 데이터베이스를 디스크가 아닌 주기억 장치에 모두 상주시키는 것이다. 반도체 메모리의 집적도가 증가하고 가격이 하락함에 따라 모든 데이터베이스를 주기억 장치에 상주시켜 트랜잭션 처리율을 높이기 위한 연구가 이루어지고 있다. 본 논문에서는 주기억 장치 데이터베이스를 기반으로 한 고성능 트랜잭션 처리 시스템을 구현하기 위하여 새로운 병행수행 제어 기법과 회복 기법 그리고 저장 구조를 제안하며, 트랜잭션의 처리량과 응답속도로 평가되는 트랜잭션 처리 시스템 성능의 개선을 목적으로 한다.

Design and Evaluation of Transaction Processing System based on Main Memory Database

Jong-Ik Shim

ABSTRACT

Nowadays, the number of database applications which need fast transaction processing are increasing. One way to improve the performance of transaction processing is to reside the whole database in main memory. As semiconductor memory becomes cheaper and chip densities increase, the research to improve transaction throughput rates of transaction processing system, using main memory databases, has begun. In this thesis, how to implement a high performance transaction processing system based on main memory databases, new concurrency control scheme, recovery scheme and storage structure is presented. The objective of the proposed schemes is to improve the transaction processing system performance measured by transaction throughput and response times.

1. 서 론

최근 들어 데이터베이스 응용분야는 다량의 데이터를 공유하기 위한 관리 차원에서 벗어나 고속의 트랜잭션 처리를 요하는 분야로 확대되어 가고 있다. 이러한 응용분야를 위한 데이터베이스와 트랜잭션 처리는 다음과 같은 요구를 만족해야 한다. 먼저, 데이터베이스는 데이터에 대한 접근(access) 시간이 매우 짧도록 설계되어야 한다. 그리고 트랜잭션은 처리율이 높아야 하고, 낮은 지연율(latency)과 엄격한 지속성(durability) 뿐만 아니라 가용성(availability)이 보장되어야 한다[1]. 주기억 장치 데이터베이스 시스템(main memory database system)은 전체 데이터베이스를 반도체 메모리(semiconductor memory)에 저장한다[2]. 기존 시스템에서는 트랜잭션의 수행 시간중 대부분이 디스크에 있는 데이터를 접근(access)하기 위하여 대기하는데 쓰여진다. 그러나 전체 데이터베이스를 주기억 장치에서 직접 접근할 수 있게 되면 기존의 디스크 기반 시스템(disk based system)에 비해 트랜잭션을 수행하기 위한 디스크 I/O가 없어지고, 트랜잭션 충돌이 줄어들기 때문에 충돌을 해결하기 위해 사용되는 블록킹(blocking)이

[†] 한서대학교 컴퓨터과학과 조교수

나 취소(abort)로 발생되는 문맥 교환(context switching)이 상당히 작아진다. 따라서, 매우 빠른 응답 시간과 높은 트랜잭션 처리 능력을 제공할 수 있다[3]. 그 결과 데이터를 주기억 장치에 저장하는 것만으로도 간단히 시스템 성능을 높일 수 있는 기술로 인식되고 있어 전체 데이터베이스를 주기억 장치에 상주시키는 시스템에 대한 연구가 활발해지기 시작하였다[4-6].

본 논문은 트랜잭션 응답속도가 빠르고 시스템 전체 처리량(throughput)이 좋으며, 빠른 회복이 가능한 주기억 장치 데이터베이스 트랜잭션 처리 시스템(Transaction Processing System based on Main Memory Databases: TPS/MMDB)을 구현하기 위하여 새로운 병행수행 제어와 회복 기법 그리고 저장 구조를 제안한다. 디스크 기반 시스템에서와는 달리 주기억 장치 데이터베이스를 갖는 트랜잭션 처리 시스템을 설계하기 위해 다음과 같은 특성들을 고려해야 한다. 첫째, 주기억 장치에 대한 접근은 디스크 접근에 비하여 속도가 매우 빠르다. 그러므로 병행수행 제어를 위해 디스크 기반 시스템에서는 별로 문제가 되지 않았던 잠금(locking)과 같은 연산이 큰 오버헤드로 작용할 수 있다. 둘째, 디스크는 영구적인 저장 구조이지만 주기억 장치는 휘발성이다. 따라서, 주기억 장치 데이터베이스에 대한 고장으로부터의 회복 문제가 매우 중요하다. 셋째, 디스크는 접근되는 동안 발췌된 데이터의 양에 관계없이 접근당 높고 고정된 비용이 요구된다. 그러나 주기억 장치는 접근 비용이 데이터의 크기에 비례한다. 결국, TPS/MMDB를 구현하기 위해서는 병행수행 제어와 고장으로부터의 회복 그리고 저장 구조 문제가 가장 중요한 연구 과제가 된다.

본 논문의 구성은 다음과 같다. 2장에서는 관련연구를 고찰하고, 3장에서 새로운 TPS/MMDB를 위한 시스템 구조와 저장 구조 그리고 병행수행제어 기법과 회복 기법을 제안한다. 4장에서는 기존의 연구 중에서 평가가 이루어진 연구 결과를 기준으로 제안된 알고리즘의 성능을 분석하고, 제안된 병행수행제어 알고리즘의 정확성을 증명한다. 그리고 5장에서 결론 및 추후 연구 과제에 대하여 기술한다.

2. 관련연구

MM-DBMS[8]는 데이터베이스 객체가 고정된 파티션으로 구성된 논리적 세그먼트에 저장하도록 되어 있다. 회복을 위해서는 회복 전용 프로세서를 가지고 있으며, 로그 버퍼를 위해 안정된 메모리를 사용한다. 병행수행 제어는 릴레이션 레벨 잠금과 튜플 레벨 잠금의 두 가지 잠금 단위(locking granule)를 사용하는 잠금 알고리즘을 사용한다. MARS[7]는 데이터베이스의 주 사본을 가지고 있는 MM(Main Memory), 그림자 영역을 위한 비휘발성 메모리의 SM(Stable Memory), 로그 버퍼를 위한 AM(Archive Memory) 디렉토리, 그리고 검사점 비트 맵으로 나뉘어진다. 지연/즉시(deferred/in-place) 갱신 기법을 사용하며, 퍼지 검사점 방식을 이용하여 휘발성 MM에 있는 갱신된 페이지들을 AM에 복사한다. 트랜잭션이 완료되면 수정된 데이터를 그림자 영역으로부터 MM에 복사하고 접근했던 릴레이션들의 잠금을 해제시키는 잠금 기반의 병행수행 제어 방식을 따른다. System M[13]은 레코드와 집합(set)들로 구성된 단순한 데이터 모델을 구현하였다. 레코드 갱신은 UNDO 로깅의 필요성을 없애기 위하여 그림자 방식을 사용하였다. 트랜잭션을 처리하기 위해 잠금 관리자는 데이터베이스 세그먼트와 레코드에 대한 잠금 기능을 제공하며, 공유(shared)와 배제(exclusive) 잠금 모드를 제공한다. 검사점을 위해 Black/white 알고리즘과 Copy-on-Update 알고리즘 등을 제안하였으며, 이를 퍼지와 동작-일치성 그리고 트랜잭션-일치성 검사점 방식에 적용하여 평가하고 있다. Starburst[9]는 IBM의 Almaden 연구소에서 개발한 프로토타입 DBMS로 기존의 디스크 기반 시스템을 확장한 것이다. 기존의 디스크 기반 시스템의 잠금 비용을 줄이기 위하여 잠금 기반 병행수행 제어를 적용한 새로운 MMM LM(MMM lock manager)을 구현하였다. Dali[1]는 데이터베이스 파일이 각자 자신의 보호 모드를 갖는 여러 개의 파티션으로 구성되어 있다. 회복 관리자는 논리적인 로깅(연산의 로깅) 뿐만 아니라 물리적인 로깅을 지원한다. 검사점 알고리즘은 ping-pong 기법을 적용하여 연속적인 검사점들이 디스크의 다른 위치에 기록되도록 한다. 병행수행 제어 관리자는 2PL 기법을 기초로 하여 구현하였다.

3. 새로운 TPS/MMDB의 제안

3.1 TPS/MMDB 시스템 구조

[그림 1]은 제안된 시스템의 구조를 보여주고 있

다. 제안된 시스템은 휘발성 표준 메모리로 구성된 주기억 장치 데이터베이스(MMDB: Main Memory Database)와 작업영역(workspace), 비휘발성의 안정된 기억 장치(stable memory), 그리고 두 종류의 디스크들로 구성된다. MMDB는 휘발성 메모리로 데이터베이스 주 사본(primary copy)을 저장하는데, 고정된 크기로 분할되어 있다. 작업영역(workspace)은 트랜잭션이 처리되는 동안 갱신된 페이지를 유지하는데 이용된다. 안정된 메모리(Stable Log Buffer)는 갱신된 페이지에 대한 로그 버퍼로 사용되어 트랜잭션을 완료할 때 디스크 I/O가 발생되지 않도록 한다. 제안된 시스템에서 사용하는 디스크들은 시스템 파손에 대비하여 데이터베이스 백업용으로 사용되는 백업 디스크(BD: Backup Disk)와 로그 정보를 저장하기 위한 로그 디스크(LD: Log Disk)가 있다. 시스템을 구성하는 소프트웨어는 관계형 데이터베이스 모델을 관리하는 저장 관리자, 트랜잭션 처리를 담당하는 트랜잭션 관리자와 병행수행 관리자, 그리고 회복 처리를 담당하는 로깅 관리자와 검사점 관리자로 구성된다.
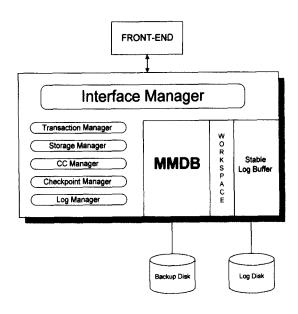
그림 1. 제안된 시스템 구조

3.2 TPS/MMDB 저장 구조

데이터베이스를 구성하는 객체인 릴레이션은 논리적인 세그먼트에 저장된다. 세그먼트는 고정된 크기의 페이지들로 이루어진 가변 길이의 데이터 구조를 가지며, 페이지는 메모리 할당과 회복의 단위로 사용된다. [그림 2]는 데이터베이스를 구성하기 위한 세그먼트와 페이지의 논리적인 구성을 나타내고 있다.

세그먼트는 헤더와 본체로 구성된다. 세그먼트 헤더는 세그먼트와 세그먼트에 속한 페이지들을 관리하기 위한 정보를 유지한다. 세그먼트 본체는 튜플들을 저장하기 위한 데이터 페이지들로 나누어진다. 데이터 페이지는 실제 레코드들이 저장되는 공간으로 [그림 3]과 같은 구조를 가지고 있다. 릴레이션을 구성하는 레코드들은 고정된 크기로 슬롯에 저장되는된다. 각 슬롯은 상태 정보 필드와 포인터 필드 형식으로 되어 있어 사용 중인 슬롯인지 아니면 빈 슬롯인지를 나타낸다. 사용되지 않고 있는 빈 슬롯들은 데이터 포인터 필드를 이용해서 리스트 형식으로 관리된다.

3.3 병행수행제어

지금까지 제안된 대부분의 주기억 장치 데이터베이스 시스템은 병행수행 제어를 위해 기존의 2PL 기법을 그대로 채택하고 있으며, 아직까지 주기억 장치 데이터베이스를 위한 병행수행 제어의 연구가 활발히 이루어지지 않고 있다[4,7,13]. 이것은 2PL이 기존의 데이터베이스 시스템에서 충분히 연구되어 왔으
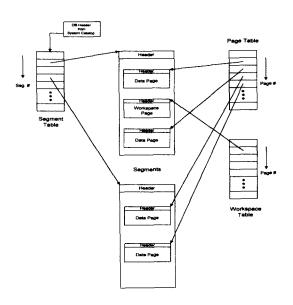
그림 2. 데이터베이스를 구성하는 세그먼트와 페이지
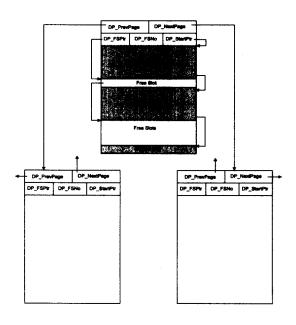
그림 3. 데이터 페이지

며 상용 데이터베이스 시스템에서 널리 사용되고 있기 때문이다. 주기억 장치는 디스크와는 달리 접근 속도가 매우 빠르기 때문에 트랜잭션의 수행이 상대적으로 빠르게 완료될 수 있다. 따라서, 주기억데이터베이스 시스템의 트랜잭션은 디스크 기반 시스템에서 보다 충돌이 훨씬 줄어들게 된다. 또한, 주기억 장치에서 하나의 페이지를 잠금하기 위한 시간은 하나의 페이지를 접근하는 시간과 비슷하다. 따라서, 잠금의 획득과 해지를 위한 동작이 디스크 기반 시스템과는 다르게 큰 오버헤드로 작용하게 된다. 이러한 특성들은 주기억 장치 데이터베이스 시스템의 효율적인 병행 수행 제어를 위해 디스크 기반 시스템에서 적용되는 기법과는 다르게 처리되어야 함을 의미한다. 낙관적인 병행수행 제어(OCC: Optimistic Concurrency Control)기법[15]은 2PL 기법과는 달리 넌블록킹 방식을 적용하고 있기 때문에 교착상태를 피할 수 있다. 또한, 낙관적인 기법은 충돌이 많이 발생하지 않는 시스템 환경에서 병행성에 대한 높은 잠재력을 가지고 있는 방식이다. 실시간 데이터베이스 시스템과 주기억 장치 데이터베이스 시스템의 목표는 다르지만 주기억 장치를 기반으로 하거나 디스크를 기반으로 이루어진 실시간 데이터베이스 시스템에서의 연구 결과들 중에서 [10]와 [11] 그리고 [12] 등은 기존의 데이터베이스 시스템에서와는 달리 낙관적인 기법이 잠금 기법보다 성능이 우수한 경우를 보이고 있다. 본 논문에서 제안하는 병행수행제어 기법은 낙관적인 방식을 기초로 한다. 충돌을 검사하기 위해 전진 검증 방식을 사용하고 있으며, 읽기단계, 검증단계 그리고 쓰기 단계로 이루어진다.

(1) 읽기 단계

이 단계에서는 트랜잭션의 연산을 수행하는데 필요한 데이터 객체들을 주기억 장치 데이터베이스로부터 지역 작업영역(workspace)으로 복사한다. 그리고 트랜잭션의 모든 갱신 연산은 이 영역에서 이루어진다. 트랜잭션의 수행이 끝나 검증 단계에 도착하여 데이터 충돌을 검사하고, 충돌이 발생되었을 경우 이것을 해결하기 위해 이용하는 데이터 객체 집합(OS)과 트랜잭션 집합(TS)내의 엔트리를 생성한다. 객체 집합 OS(T)은 읽기 단계에 있는 트랜잭션 T가 쓰기 연산을 하는 데이터 객체들의 집합이고, 트랜잭션 집합 TS(Q)는 데이터 객체 Q에 읽기 연산을 하는 활동 중인 트랜잭션들의 집합이다.

(2) 검증단계

수행중인 모든 트랜잭션들은 아무런 제약 없이 병렬로 수행되다가 검증 단계에 도달하게 된다. 검증단계에 도착하면 데이터 충돌이 있는지를 검사한다. 검증하고 있는 트랜잭션 T_v가 쓰기 연산을 하는 객체 집합 OS(T_v)내에 항목 Q를 가지고 있고, 이 항목 Q에 읽기 연산을 하는 활동중인 트랜잭션들의 집합 TS(Q)가 존재하면 충돌이 발생한 것이다. 검증하고 있는 트랜잭션 T_v의 충돌 빈도를 conflictNo(T_v), 충돌한 트랜잭션 T_c의 충돌 빈도를 conflictNo(T_c) 라고 하자. 이때, conflictNo(T_v)가 모든 conflictNo(T_c)보다 크면 검증 단계에 있는 T_v를 쓰기 단계로 보내어 완료가 되도록 하고, 활동중인 충돌한 트랜잭션 T_c를 모두 재시작시킨다. 반대로 conflictNo(T_v)가 어떤 conflictNo(T_c) 보다 작으면 검증하고 있는 트랜잭션 T_v를 취소하고, T_c를 계속 수행하도록 한다. 충돌빈도가 많아 긴 트랜잭션이거나 쓰기 위주의 트랜잭션일 가능성이 높은 T에게 완료 우선권을 주는 것이다. 만약, 이러한 T가 검증 단계에서 취소된다면, 재시작됨으로써 자원의 낭비가 심해지고 충돌의 발생 가능성이 지속적으로 높아지게 된다.

각 트랜잭션에 대한 충돌 빈도 conflictNo(T)의 계

산은 읽기 단계에서 생성된 자신의 OS(T)와 TS(Q)의 정보를 이용한다. 트랜잭션 T에 의해 쓰기 연산이 이루어지는 데이터 객체 Q_i(i=1,2,...m)가 존재하면, Q_i에 읽기 연산을 하는 트랜잭션들이 존재하는지를 TS(Q_i)에서 찾는다. 이때, 모든 Q_i에 대한 TS(Q_i)의 수가 트랜잭션 T에 대한 충돌 빈도가 된다.

(3) 기아현상

낙관적인 병행수행 제어 기법은 일반적으로 긴 트랜잭션에서 충돌로 인하여 트랜잭션들이 계속해서 재시작하게 되는 트랜잭션 기아 현상(starvation problem)이 발생될 수 있다[10]. 그러나 본 논문에서 제안한 낙관적인 병행수행 제어 기법은 충돌 가능성이 많은 긴 트랜잭션에 우선권을 부여하는 정책을 따르기 때문에 충돌이 비교적 작은 트랜잭션들이 반복적으로 취소되는 현상이 일어날 수 있다. 이러한 기아 현상을 해결하기 위해 트랜잭션 T가 취소되어 재시작할 때에는 재시작 횟수에 대한 정보를 특정 테이블(RN)에 유지한다. 그리고 재시작 횟수는 트랜잭션이 검증 단계에 도착하여 충돌 빈도를 비교할 때 반영된다. 재시작되는 경우가 많아질수록 그 횟수에 따라 가중치(wf)가 부여되고 트랜잭션의 완료될 가능성이 점차 높아지게 되어 충돌이 작은 트랜잭션에 대한 기아 현상을 해결하게 된다. 이때, 재시작 빈도에 주어지는 가중치, wf는 다음과 같이 구해진다. [알고리즘 1]은 제안된 병행수행 알고리즘을 나타내고 있다.

$$C_{diff} = \sum_{c=1}^{n} ConflictNo(T_c) / ConflictNo(T_v) - ConflictNo(T_v) \quad 1$$

$$wf = C_{diff} / ConflictNo(T_v)$$

(단, c≠v)

[알고리즘1] 제안된 병행수행 제어 알고리즘

```
begin
  valid := true;
  foreach Q in OS(Tv)
  begin
    foreach Tc in TS(Q)
    begin
      if conflictNo(Tv)+wf*RN[Tv]<conflictNo(Tc)
          then begin
              restart(Tv);
              adjust(Tv);
              valid := false;
              exit loop;
          end
        end-if
    end
    if not valid
      then exit loop;
    end-if
  end
  if not restarted Tv
    then begin
      foreach Tc in TS(Q)
      begin
        restart(Tc);
        adjust(Tc);
      end
      execute write_phase(Tv);
    end
  end-if
end
```

(4) 쓰기단계

트랜잭션이 검증 단계를 거쳐 쓰기 단계로 넘어오면 완료(commit)된 것으로 본다. 쓰기 단계에서는 완료된 트랜잭션의 작업영역에 있는 수정된 페이지들을 회복을 위하여 안정된 로그 버퍼에 기록하고, 데이터베이스에 반영한다. 제안된 알고리즘에서는 직렬성을 보장하기 위하여 검증 단계와 쓰기 단계를 하나의 임계 구역(critical section)에서 실행하도록 한다.

3.4 회복 기법

(1) 로깅과 로그처리

제안된 구조에서는 트랜잭션의 완료 처리를 위하여 낙관적인 병행수행 제어 기법을 따르고, 로그 버퍼를 안정된 메모리에 유지하여 로깅이 이루어진다. 따라서, 트랜잭션 수행 도중에는 갱신 연산에 의해서 변경된 어떠한 페이지들도 주기억 장치에 반영되지 않는다. 그리고 수행 도중 여러 가지 원인에 의해 취소된 트랜잭션의 부분적 연산 결과는 작업영역의 해제와 함께 로그에 반영되지 않고 완전히 제거된다. 결국, 로그는 완료된 트랜잭션의 AFIM(After Image)들만으로 구성된다. 이것은 지역 작업영역이 그림자 영역처럼 작용하여 BFIM(Before Image)를 로그 버퍼에 기록할 필요가 없기 때문에 가능하다. 제안된 구조에서는 AFIM들이 완료된 트랜잭션 순서

에 따라 로그에 연속적인 형태로 기록하게 한다. 이와 같은 형태로 로깅이 이루어지도록 보장하기 위해서 완료된 트랜잭션들이 로깅을 요구할 때 로그 페이지를 그 트랜잭션의 AFIM 크기만큼 로그 버퍼에 미리 할당하는 방법을 사용한다.

고장이 발생되었을 때, LBC를 완료된 검사점의 시작 지점이라 하고, BC를 수행 중인 검사점의 시작 지점이라 하자. 현재 가장 널리 사용되고 있는 로깅 방식인 WAL 프로토콜은 로그 레코드를 먼저 로그 버퍼에 기록한 다음 갱신된 페이지들을 데이터베이스에 반영한다. 따라서, BC(LBC) 레코드는 로그 레코드를 로깅하는 시점과 데이터베이스를 MMDB로 갱신하는 시점 사이에 기록될 가능성이 있다. 이러한 경우, MMDB가 파손되었을 때 LBC에서부터 로그를 처리하게 된다면, LBC 시점에서 수행 중이던 트랜잭션들의 데이터베이스를 일치된 상태로 복구할 수 없게 된다. 따라서, 로그 처리의 시작 지점은 LBC에서 진행 중이던 트랜잭션들 중에서 가장 일찍 시작한 트랜잭션의 시작 지점(Oldest-BT)이 되어야 한다. 즉, 일반적인 직접 갱신 방식에서 고장 발생 후 로그 처리를 하기 위한 과정은 Oldest-BT를 기준으로 하여 분석, UNDO 그리고 REDO의 세 단계로 이루어져야 한다. 그리고 UNDO와 REDO 처리를 효율적으로 처리하기 위해 회복 관리자는 완료된 트랜잭션들의 리스트(CTL: committed transaction list)와 취소된 트랜잭션들의 리스트(ATL: aborted transaction list)를 만들어야 한다.

고장이 발생되었을 때 로그 처리를 하기 위하여 로그 디스크에 저장된 로그를 가지고 이루어지는 이러한 작업은 회복 과정에 커다란 오버헤드로 작용한다. 가장 이상적으로 빠르게 로그를 처리하기 위해서는 로그를 검사하는 분석 단계와 UNDO 연산 없이 곧 바로 REDO 연산을 하게 하는 것이다. 또한, 이러한 REDO 연산은 Oldest-BT에서부터가 아닌 LBC에서부터 이루어지게 하는 것이 가장 효율적이다. 이렇게 로그 처리가 이루어지도록 하기 위해 새로이 제안하는 퍼지 검사점 알고리즘은 LIC(Lost Image Counter) 레코드를 이용한다. LIC 레코드는 퍼지 검사점이 로그에 BC 레코드를 기록한 후 BC 레코드 이전에 로깅한 트랜잭션들이 변경된 데이터를 MMDB에 모두 갱신하였는지를 확인하기 위한 두 개의 LIC 변수와 0과 1의 값을 가질 수 있는 curr_cnt_bit (current counter bit)로 이루어진다.

트랜잭션의 수행이 완료되어 로그 버퍼에 AFIM를 기록할 때, LIC[curr_cnt_bit] 값을 하나 증가시키고, MMDB에 갱신하고 나면 다시 감소시킨다. LIC[] 값을 확인하여 그 값이 0이면 AFIM를 로깅한 모든 트랜잭션들이 MMDB에도 갱신한 상태를 나타내기 때문에 검사점을 실행할 수 있는 조건이 된다. LIC[] 값이 0이 아니면, BC 이전에 AFIM를 로깅하였으나 MMDB에는 아직 갱신하지 않은 트랜잭션이 존재하는 것이다. 따라서, BC 이전에 로깅한 모든 트랜잭션의 갱신이 이루질 때까지 즉, LIC[]가 0이 될 때까지 기다린다. 이때, 트랜잭션들은 아무런 동기 과정 없이 정상적인 연산을 수행할 수 있으며, 로그 버퍼에 로깅을 계속 수행할 수 있다. 그 이유는 LIC 변수를 두 개로 유지하면서 새로운 검사점이 시작될 때마다 번갈아 가며 사용하기 때문이다. [알고리즘 2]는 앞 절에서 기술한 로깅 방식과 LIC 레코드를 이용하여 검사점이 이루어졌을 때의 로그 처리 알고리즘을 보여주고 있다.

[알고리즘 2] 제안된 로그처리 알고리즘

```
begin
   read log starting from LBC
   while end of the log is not reach
   begin
      foreach log record
         copy AFIM to workspace
   end
end
```

(2) 검사점

[알고리즘 3]은 제안된 퍼지 검사점 과정을 보여주고 있다. 검사점을 수행하기 위한 준비 과정으로 로그에 BC레코드를 기록하고, curr_cnt_bit의 값을 바꾸어(현재의 값이 0일 경우에는 1로, 1일 경우에는 0으로) 현재 수행중인 트랜잭션이 로깅을 지속적으로 진행할 수 있도록 한다. 그리고 BC 레코드를 로그에 기록하기 전까지 사용된 LIC[curr_cnt_bit]의 값이 0이 되는 시점에서 검사점을 시작한다. 검사점은 지속적으로 수행되도록 하고, 주기억 장치 데이터베이스 내의 페이지들 중 더티 페이지(dirty page)들만을 백업한다. 안정된 메모리내의 페이지 비트맵을 조사하여 해당 페이지에 대한 더티 비트가 세트되어

있으면 더티 비트맵의 대응되는 비트를 리셋시키고, 그 페이지를 백업 디스크에 복사한다. 이러한 모든 과정은 로깅이나 트랜잭션 수행과는 독립적으로 수행되며, 동기성 검사점과 달리 페이지들에 대한 어떠한 잠금이나 트랜잭션의 중단 없이 이루어진다.

[알고리즘 3] 제안된 퍼지 검사점 알고리즘

```
begin
  lock LIC record
  write the BC record in the log
  set pre_cnt_bit to curr_cnt_bit
  switch curr_cnt_bit
  unlock LIC record
  while LIC[pre_cnt_bit] is not 0
  begin
  end
  while there is a database page unchecked
  begin
    if the page is dirty
      then begin
        reset the bit in the dirty bitmap
        copy data to the I/O buffer
        request I/O to flush this page to its location
          in BD
      end
  end
  write the EC record in the log
end
```

(3) 트랜잭션처리

트랜잭션의 수행중 갱신 연산이 존재하면 데이터베이스로부터 해당 페이지를 작업영역/그림자 영역에 복사하여 갱신이 이루어진다. 트랜잭션의 수행이 완료되면 검증 단계에서 충돌을 검사한다. 이때 충돌이 발생하지 않으면 완료 처리 단계에 들어간다. 충돌이 발생하였을 경우에는 제안된 충돌 해결 방법에 의해서 검증하고 있는 트랜잭션의 상태를 결정한다. 취소가 되는 경우에는 트랜잭션 고장의 경우와 마찬가지로 처리된다. 즉, 별도의 UNDO 연산 과정 없이 단지 작업영역을 해제시킨다. 트랜잭션의 완료 처리 과정은 먼저 회복을 대비하여 모든 AFIM들을 로그에 기록하는 것으로 이루어진다. 로그는 안정된 메모리 내에 유지되고 있기 때문에 트랜잭션이 완료되기 위하여 로그 레코드가 디스크에 쓰여지기를 기다릴 필요가 없다. 그 다음, 작업영역에 있는 갱신된 페이지들을 주기억 장치 데이터베이스에 반영시킨다. 그리고 갱신된 페이지의 더티 비트맵을 세트시킨다. 이 비트는 검사점 수행시 BD에 백업시킬 것인지에 대한 검사가 이루어질 때 사용된다. 이때, AFIM를 안정된 로그 버퍼에 쓰고 나면 LIC[curr_cnt_bit]를 하나 증가시키고, 작업영역에 있는 변경된 데이터를 MMDB에 갱신하고 난 후에는 LIC[pre_cnt_bit]를 감소시켜서 퍼지 검사점이 시작하기 위한 정보로 사용될 수 있도록 한다. [알고리즘 4]는 이러한 낙관적인 병행수행 제어와 회복 기법을 통합한 트랜잭션 처리 과정을 보여주고 있다.

[알고리즘 4] 제안된 트랜잭션 처리 알고리즘

```
begin [Begin-Transaction]
  while transaction T is not reached at commit point
  begin
    if there is an update operation
        copy request pages from MMDB to workspace
    compute database
  end
  validate transaction T
  if transaction accepted
    then begin
      lock LIC record
      write all AFIM to log
      LIC[curr_cnt_bit]++
      set pre_cnt_bit to curr_cnt_bit
      unlock LIC record
      copy all updated pages in workspace to MMDB
      set every dirty page bitmap of updated page
        to 1
      lock LIC record
      LIC[pre_cnt_bit]--
      unlock LIC record
    end
  else
      abort transaction T
  free workspace of transaction T
end [End-Transaction]
```

4. 성능 평가

본 절에서는 제안한 회복 기법의 성능을 분석하고 제안된 병행수행 제어 기법의 정확성을 증명한다. 새로이 제안한 기법은 회복 기법에 관한 연구들 중에서 성능 평가 결과를 제시한 분할 방식[4]과 기존의 퍼지 검사점 방식을 기초한 회복 기법을 가지고 비교한다. 객관적인 성능 비교를 위해 [14]에서 사용한 분석 기법으로 회복 수행에 필요한 비용을 계산한다. 분할 방식과 기존의 퍼지 검사점 방식을 기초한 회복 기법

의 비용은 [4]과 [14]을 참조하여 그 결과만 인용하여 비교하였다. [표 1]은 성능 분석에 사용되는 파라메터들을 보여주고 있다.

4.1 로깅과 검사점 비용

로깅과 검사점 수행을 위한 공간과 시간의 비용을 얻기 위해 로그의 총 크기를 계산한다. 분할기법의 총 로그 크기는

$S_{partlog}=(S_{BC}+S_{EC})+(N_{tran}*(S_{BT}+((1-P_{abort})*S_{ET})))+(S_{BT}*N_{part}*N_{tran})+(N_{BFIM}*S_{BFIM})+(N_{AFIM}*S_{AFIM})+(N_{locchkpt}*(S_{BC}+S_{EC}))$ 이다.

퍼지 검사점 로그에 대한 총 크기는

$S_{fuzzlog}=(S_{BC}+S_{EC})+((S_{BT}+(S_{ET}*(1-P_{abort})))*N_{tran})+(N_{BFIM}*S_{BFIM})+(N_{AFIM}*S_{AFIM})$ 이다.

새로 제안된 회복 기법은 그림자 기법의 성질을 가지고 있기 때문에 BFIM를 가질 필요가 없으며, 트랜잭션의 시작이나 종료 혹은 취소를 나타내기 위한 레코드들이 필요하지 않다. 따라서, 총 로그의 크기는 다음과 같다.

$S_{newlog} = (S_{BC}+S_{EC})+(N_{AFIM}*S_{AFIM})$.

새로운 기법은 기존의 퍼지 검사점 기법에 비해 $(S_{BT}+S_{ET}*(1-P_{abort})))*N_{tran})+(N_{BFIM}*S_{BFIM})$ 만큼 로그의 크기가 작아지며, 분할 기법에 비해 $(S_{BT}*N_{part}*N_{tran})+(N_{locchkpt}*(S_{BC}+S_{EC}))-(N_{BFIM}*S_{BFIM})+(N_{tran}*(S_{BT}+((1-P_{abort})*S_{ET})))$ 만큼의 크기가 작은 것을 알 수 있다.

로그의 크기가 작으면 로깅하는데 걸리는 시간이 짧아지게 되기 때문에 정상적인 트랜잭션의 수행 시간을 단축시킬 수 있다. 그리고, 로그의 크기가 작으면 로그 버퍼내의 로그 레코드들을 로그 디스크로 저장하기 위한 검사점 수행 시간이 짧다는 것을 의미하여 고장을 대비한 회복 과정이 정상적인 트랜잭션 수행에 미치는 영향이 작다는 것을 말한다.

4.2 회복 처리 분석 비용

고장이 발생한 후 회복 처리를 위하여 트랜잭션의

표 1. 성능분석 파라메터

Paramete	Meaning	Default	Range
N_{tran}	Number of transactions		200-1000
N_{BFIM}	Number of BFIM records	calc	
N_{AFIM}	Number of AFIM records	calc	
N_{part}	Number of partitions	5	
$Nl_{occhkpt}$	Number of local checkpoints	calc	
P_{upd}	Probability that an operation is update	0.1	0.1-0.4
P_{abort}	Probability that a transaction does not commit	0.03	
S_{BT}	Size of BT record	2	
S_{ET}	Size of ET record	2	
S_{BC}	Size of BC record	2	
S_{EC}	Size of EC record	2	
S_{BFIM}	Size of BFIM record	5	
S_{AFIM}	Size of AFIM record	5	
S_{tran}	Number of operations in transaction	5	5-20
$S_{partlog}$	Total size of log for partition checkpoint	calc	
$S_{fuzzlog}$	Total size of for fuzzy checkpoint	calc	
S_{newlog}	Total size of for new checkpoint	calc	
$S_{parttran}$	Number of log words to identify transaction status for partition scheme	calc	
$S_{fuzztran}$	Number of log words to dentify transaction status for fuzzy scheme	calc	
$S_{newtran}$	Number of log words to dentify transaction status for new scheme	calc	
$S_{partrecv}$	Number of log words to perform redo for patition scheme	calc	
$S_{fuzzrecv}$	Number of log words to perform redo for fuzzy scheme	calc	
$S_{newrecv}$	Number of log words to perform redo for new scheme	calc	

상태를 분석하는데 걸리는 시간을 비교해 보기로 한다.기존의 퍼지 기법에서 분석 비용은 다음과 같은 총 로그의 크기에 비례하게 된다.

$S_{fuzztran} = S_{fuzzlog}$.

분할 기법에서는 전역 로그만 검사하면 된다. 즉, 다음과 같은 전역 로그의 크기에 비례한다.

$S_{parttran}=(S_{BC}+S_{EC})+((S_{BT}+((1-P_{abort})*S_{ET}))*N_{tran})$.

새로이 제안한 회복 기법은 REDO 처리를 위하여 전체 로그를 검사할 필요가 없이 LBC 위치를 알아내기 위해 S_{BC}와 S_{EC}만 읽으면 된다. 따라서,

$S_{newtran} = (S_{BC}+S_{EC})$ 이다.

새로이 제안된 알고리즘은 로그 처리를 위하여 별도의 준비 과정이 필요 없으며, 트랜잭션의 양에 전혀 상관없이 일정하다는 것을 알 수 있다. 그리고, 분할 기법은 퍼지 검사점에 비해서 다음과 같은 비용이 절감됨을 알 수 있다.

$(N_{BFIM}*S_{BFIM})+(N_{AFIM}*S_{AFIM})$.

이러한 비용은 분할의 수와 상관없으며 트랜잭션의 연산수와 관계된다. 이 결과는 트랜잭션당 연산의 수를 20으로 하여 분석한 결과이다. 4.1의 결과와 비교하여 본다면, 10개의 분할에 따른 오버헤드는 20개 연산을 갖는 트랜잭션의 경우에 의해 보상될 수 있음을 알 수 있다. 그러나 새로운 회복 기법은 트랜잭션의 성질에 상관없이 비용이 일정하며 준비 과정 없이 곧 바로 회복을 위한 로그 처리를 할 수 있다.

4.3 로그 처리 비용

회복 처리는 UNDO 연산과 REDO 연산 시간으로 평가된다. 먼저, UNDO 연산에 대해서는 분할 기법과 기존 퍼지 기법의 비용이 동일하다. UNDO 연산은 취소된 모든 트랜잭션과 활동 중이던 트랜잭션에 대하여 이루어진다. 그러나 새로이 제안한 기법에서는 낙관적인 병행수행 제어의 특성으로 그림자 방식 효과를 얻기 때문에 UNDO 연산이 필요하지 않아 회복 처리 시간이 그 만큼 줄어들게 된다. 이 실험에서는 객관적인 성능 비교를 위해 REDO를 위한 로그 처리 비용만을 분석해 본다.

분할 기법이 REDO 단계에서 사용되는 총 로그의 크기는 다음과 같다.

$$S_{partrecov}=(2/(N_{part}+1))*((S_{BT}*N_{part}*N_{tran})+(N_{BFIM}*S_{BFIM})+(N_{AFIM}*S_{AFIM})+(N_{part}*(S_{BC}+S_{EC}))).$$

기존의 퍼지 기법에서 REDO 처리를 하기 위한 로그 레코드의 크기는 다음과 같이 전체 로그가 된다.

$S_{fuzzrecov} = S_{fuzzlog}$.

새로이 제안한 기법에서 REDO 처리를 하기 위한 총 로그의 크기도 다음과 같이 전체 로그가 된다.

$S_{newrecov} = S_{newlog}$.

분할 기법은 기존의 검사점 기법에 비해($S_{fuzzlog}-S_{partrecov}$)만큼의 비용이 절감되어 약 56%의 비용이 줄어든다. 그러나 분할 기법은 새로운 회복 기법에 비해 회복 처리해야 될 로그의 양이 두 배 이상 크다. 이것은 분할 기법이 REDO 연산을 하기 위해 지역 로그를 검사해야 하며, REDO 처리만 수행하더라도 BFIM가 포함된 전체 로그를 읽어야 하기 때문이다. 그러나, 제안된 기법은 BFIM를 로그에 기록하지 않기 때문에 로그의 크기가 작아져 REDO 연산이 빠르게 처리되며, UNDO 처리가 필요 없다. 분할 기법은 제안된 기법과는 달리 UNDO 처리를 해야 하는데, 이 과정도 역시 AFIM와 BFIM를 모두 포함하는 전체 로그를 디스크로부터 읽어 처리해야 한다. 제안된 기법은 분할 기법에 비해 로그 처리 비용이 약 50%, 기존의 퍼지 기법에 비해 약 70% 정도 작은 것을 알 수 있다.

4.4 병행수행제어 알고리즘의 정확성 증명

알고리즘의 정확성은 제안된 알고리즘(OCC-CF)에 의해 생성된 모든 수행기록(history)이 주기(cycle)를 갖지 않는 직렬성 그래프(serialization graph)로 표현되어 직렬성이 보장되는 것으로 증명한다[16]. H를 수행기록이라 하고 SG(H)를 H에 대한 직렬성 그래프라고 하자. 수행기록 H가 직렬 가능함을 보이기 위해서는 SG(H)가 비주기(acyclic)임을 증명하면 된다.

보조정리: T_i와 T_j를 새로운 알고리즘에 위해서 생성된 수행기록, H에서의 완료된 두 트랜잭션이라고 하자. 만약, SG(H)상에 간선 $T_i \rightarrow T_j$가 존재하면 $TS(T_i) < TS(T_j)$이다.

경우 1: $r_i[x] \rightarrow w_j[x]$

OCC-CF에서 T_i가 검증 단계에 도착하였을 때 T_i가 x에 대한 쓰기 연산을 하지 않으면 OS(T_i)내에 데이터 항목 x가 존재하지 않는다. 이때에는 검증 단계에서 충돌이 발생하지 않는 경우이기 때문에 트랜잭션 T_i가 먼저 완료된다. 따라서, TS(T_i) < TS(T_j)가 된다.

경우 2: $w_i[x] \rightarrow r_j[x]$

OCC-CF에서 검증하고 있는 트랜잭션 T_i의 conflictNo(T_i)가 충돌한 트랜잭션 T_j의 conflictNo(T_j)보다 크면 T_j는 읽기 단계에서 반드시 취소되기 때문에 항상 TS(T_i) < TS(T_j)가 된다.

경우 3: $w_i[x] \rightarrow w_j[x]$

OCC-CF에서는 검증 단계와 쓰기 단계가 동일한 임계 구역에서 수행이 되기 때문에 항상 먼저 도착한 트랜잭션이 먼저 쓰기를 하고 완료된다. 따라서, TS(T_i) < TS(T_j)가 된다.

정리: 제안된 OCC-CF 알고리즘에 의해 생성된 모든 수행기록은 직렬가능(serializable)하다.

증명: H를 제안한 알고리즘에 의해 생성된 수행기록이라고 하고, SG(H)가 주기 $T_1 \rightarrow T_2 \rightarrow ... \rightarrow T_n \rightarrow T_1$을 갖는다고 가정하자. 보조정리에 의해서 TS(T_1) < TS(T_2) < ... < TS(T_n) < TS(T_1) 을 얻게 되고, 결과적으로 TS(T_1) < TS(T_1) 가 된다. 그러나 이것은 모순이다. 따라서, SG(H)에는 주기가 존재하지 않기 때문에 OCC_CF 알고리즘은 직렬 가능한 수행기록만을 생성한다.

5. 결 론

본 논문에서는 주기억 장치 데이터베이스를 기반으로 한 고성능 트랜잭션 처리 시스템을 구현하기 위하여 객체의 저장 및 접근 등을 위한 저장 구조와 병행수행 제어 기법, 그리고 시스템 고장을 대비한 회복 기법을 제안하였다. 제안된 병행수행제어 기법은 데이터 충돌이 많지 않은 데이터베이스 환경에 적합하며, 교착상태가 발생하지 않는다. 검증단계에서 충돌을 해결하기 위해 충동한 트랜잭션들간의 충돌 정보를 이용하여 시스템의 성능을 향상시키고자 하였다. 그리고 자주 취소되는 트랜잭션들의 기아현상을 방지하기 위하여 취소된 횟수를 가중치로 반영하는 알고리즘을 제안하였다. 또한, 본 논문에서는 새로운 회복 기준을 설정하여 회복 기법을 제안하였다. 제안된 퍼지 검사점 알고리즘은 LIC 레코드를 이용하여 WAL 프로토콜의 단점을 개선하였으며, 병행수행 제어를 위한 작업영역이 그림자 영역 효과를 나타낸다. 따라서, 별도의 그림자 영역을 설정하지 않아도 되기 때문에 메모리 공간을 효율적으로 사용하게 된다. 또한, UNDO 연산이 간단해지고 AFIM만을 로깅함으로써 로그의 양이 줄어들어 로그 처리 시간이 단축되기 때문에 빠른 회복을 보장하게 된다. 그리고 완료된 트랜잭션의 리스트와 취소된 트랜잭션의 리스트를 생성하기 위해 로그 디스크에 저장된 로그를 역방향으로 읽는 분석 과정의 오버헤드가 없어 회복 시간을 단축시킬 수 있다. 향후 연구과제는 제안된 병행수행 제어 기법에서 데이터베이스의 접근 단위(granularity)가 트랜잭션 처리에 미치는 영향과 LIC 레코드를 이용한 회복 기법이 트랜잭션 처리에 미치는 영향을 연구하여 하나의 LIC 레코드를 많은 트랜잭션들이 동시에 접근할 때 발생되는 오버헤드를 분석하여 개선할 수 있는 방안이 요구된다.

참 고 문 헌

[1] H. V. Jagadish, et al, "Dali: A High Performance Main Memory Storage Manager," Proc. of the 20th VLDB Conf., pp.48-59, 1994.

[2] Salem, Kenneth., and H. Garcia-Molina, "Checkpointing Memory-Resident Databases," In Proc. of Intl, Conf. on Data Engineering, pp.452-462, 1989.

[3] H. Garcia-Molina and K. Salem, "Main Memory Database Systems: An Overview," Transaction on Knowledge and Data Engineering, IEEE, Vol.4, No.6, pp.509-516, 1992.

[4] Xi Li, Margaret H. Eich, and V. John Joseph, "Checkpoint and Recovery in Partitioned Main Memory Databases," IASTED International Conference on Intelligent Information Management Systems, Washington DC., June 1995.

[5] H. V. Jagadish, Silberschatz, Avi., and S. Sudarshan, "Recovering from Main Memory

Lapses," In Proc. of VLDB Conf., pp.391-404, 1993.

[6] Margaret H. Eich, Sun Wei-Li, "Nonvolatile Main Memory: An overview of Alternatives," Southern Methodist University, Technical Report 88-CSE-6, Jan, 1988.

[7] Margaret H., Eich, "MARS: The Design of A Main Memory Database Machine," Proc. of the 1987 International workshop on Database Machines, Oct., 1987.

[8] T.J. Lehman, Design and Performance Evaluation of a Main Memory Relational Database System, Ph.D. thesis, University of Wisconsin, Madison, 1986.

[9] T. J. Lehman, et. al, "An Evaluation of Starburst's Memory Resident Storage Component," IEEE Trans. on Knowledge and Data Engineering, Vol. 4, No. 6, pp. 555-566, December 1992.

[10] J. Huang, J. A. Stankovic, K. Ramamritham, and D. Towley, "Experimental Evaluation of Real-time Optimistic Concurrency Control Schemes," Proceedings of the 17th VLDB Conf., 1991.

[11] Jayant R. Haritsa, Michael J. Carey, and Miron Livny, "Dynamic Real-Time Optimistic Concurrency Control," 11th IEEE Real-Time Systems Symposium, pp.94-103, 1990.

[12] Le Gruenwal, Sichen Liu, "A Performance Study of Concurrency Control in A Real-Time Main Memory Database System," Sigmod Record, Vol. 22, No. 4, pp. 33-44, December 1993.

[13] K. Salem and H. Garcia-Molina, "System M: A Transaction Processing Testbed for Memory Resident Data," IEEE Transactions on Knowledge and Data Engineering, Vol.2, pp. 161-172, Mar. 1990.

[14] Li, Xi., et al., "Partition Checkpointing in Main Memory Databases," Technical Report, 93-CSE-23, Dept. of Computer Science and Eng., southern Methodist Univ., Oct. 1993.

[15] H. T. Kung, and J. T. Robinson, "On Optimistic Methods for Concurrency Control," ACM Transactions on Database Systems, Vol.6, No.2, pp. 213-226, 1981.

[16] P. A. Bernstein, V. Hadzilacos, and N. Goodman, *Conncurrency Control and Recovery in Database Systems*, Addison-Wesley, 1987.

심 종 익

인하대학교 전자계산공학과에서 석사와 공학박사학위를 취득하였다. 1986년부터 1993년까지 LG연구소 선임연구원으로 일하였으며, 1994년부터 한서대학교 컴퓨터과학과 조교수로 재직중이다. 주요 관심분야는 데이터베이스 분야이다.